Bucharest University of Economic Studies
The Faculty of Economic Cybernetics, Statistics and Informatics
IT&C Security Master

DISSERTATION THESIS

Coordinator
Lecturer Ph.D. Andrei TOMA

Graduate
Mihail Irinel PREDA

Bucharest
2022

Bucharest University of Economic Studies
The Faculty of Economic Cybernetics, Statistics and Informatics
IT&C Security Master

Developing a Fully Homomorphic Encryption WebAssembly module

DISSERTATION THESIS

Coordinator
Lecturer Ph.D. Andrei TOMA

Graduate
Mihail Irinel PREDA

Bucharest
2022

Statement regarding the originality of the content

I hereby declare that the results presented in this paper are entirely the result of my own creation unless reference is made to the results of other authors. I confirm that any material used from other sources (magazines, books, articles, and Internet sites) is clearly referenced in the paper and is indicated in the bibliographic reference list.

Summary

1. Introduction

From the beginning of the mankind knowledge and information were crucial in the survival of humans as a species, as well as for the rapid development of the society that we know today.

In antiquity, the first informational tool had a massive impact, was the mail system developed by Persians. Made of a chain of horses strategically put from 60 to 100 km, the postal mail was a very clever and somewhat rapid way of deliver information, especially for their generals that relied heavily on intelligence on the combat field.

Despite its barbarism and cruelty, the medieval period sought the invention of the printing press by Johannes Gutenberg which created the means for all people, irrespective of their social status to have access to knowledge and information. This has led to an increase in all aspects of society, ranging from household economies, cultural heritage and a better management of political and military organizations.

After two centuries, due to Gutenberg's printing press, a plethora of new ideas and concepts have emerged from ordinary people that traditionally could not have access to the information. This period is known as the enlightenment era and it this characterized by the vast amount of educated people who introduced to the world many economical and philosophical ideas as well as some inventions that are still used and valid today. Honorable mentions include bifocal lenses and odometer invented by Benjamin Franklin, as well as the thermometer invented by Daniel Gabriel Fahrenheit.

Modern era, or the age that has alternated between industrial revolutions and great wars, has solidified the idea that the 'information is power' and it is a valuable asset at any given time. This period is marked of a blossom of inventions and among them the humanity greatest one yet: the computer. The PC, or personal computer has had an immense impact on all aspects of the human life, continuously integrating in our life. Although it has revolutionized the way of life and often the thinking of some aspects, the PC resolved a problem, at the expanse of introducing another: the computation of large amounts of data.

Contemporary period or Information Age tries to solve that problem by introducing new ways of computing such large volumes of data. According to World Economic forum report from 2018 [1], by the year 2025 the revenue from information extracted from personal data will exceed 1.7 trillion dollars, making it the first time in one and half century when the oil industry will step down from the most lucrative business, with 1.6 trillion dollars.

In this age, information it is seen as a product of data [2]. Valuable data is processed in information and further into knowledge. For this to happen you need a large volume of data to

be processed, while keeping in mind the legal issues that arise from dealing with personal data [3], as well as the environmental impact of it.

The emergence of industry 4.0, that is centered around data collection, integration of solutions in every aspect of human activity and the processing of it has changed the society as well as the technologic landscape. This new phase of industry has brought numerous security and scalability concerns. The increase in data processing and the privacy nature of it mark the need of using secure means of handling this sensible information. According to Patil et al. [4], refers to the large volume of data collected from social networks, smart phones, IoT devices as well as other sources that create huge volumes of data. The majority of this data must be processed, while respecting privacy concerns with utmost seriousness. While most of the critical information, such as bank account, credit card or health information are being protected with well-established cryptographic algorithms such as AES [5] or Keccak, the other types of information are much more vulnerable. The computational effort and overhead of those algorithms will add to it, compared with the value of information that the data has, corroborated with the risk of breaching GDPR or other privacy laws translates into little to no incentives to process this data. This is where homomorphic encryption [6] comes in.

Homomorphic encryption refers to a set of computations that can be done on an already encrypted data without the need of decrypting it, allowing to preserve the privacy of data. It uses arithmetic circuits that focus on multiplication and addition, allowing the user to multiply and add numbers while encrypted. Being a relatively new technology on the cryptographic scene, the homomorphic encryptions have to go a long way to improve their efficiency, capabilities and the supported platforms.

Appeared in 1978 as a concept and later becoming a full-fledged branch of the cryptography, homomorphic encryption enables outsourcing of many types of computations that previously had to be kept in-house due to confidentiality constraints, including financial processing, health-data processing, and genome research. Despite the computational cost, the outsourcing perspective makes homomorphic encryption has an economic advantage on the long run.

Currently, numerous libraires have implemented most of encryption schemes for x86 architecture, but the web applications cannot use them directly. Although there are attempts to improve situation on the web, very few libraries are easy to use out of the box. Most of the time in development phase, you need to recompile the whole library which translates in hours and hours of waisted time. Although some of them are well-thought and have generic implementations, the APIs is bloated and for each operation that you do you, must input specific parameters and constants that are very hard to fully understand without spending several hours throughout the documentation. The perspective of cloud computing [7, 8, 9] and the ease of use, which always has characterized the web are the main incentives for making this module.

2. Fully Homomorphic Encryption

The concept of Homomorphic encryption was initially introduced by Rivest et al. [10] in 1978, by proposing RSA public key encryption algorithm, a multiplication homomorphism and the security of their scheme is based on factorization of integers. They were followed by ElGamal encryption scheme [11], based on multiplication homomorphism and Paillier encryption scheme [12] with the addition of homomorphism property which was an encryption scheme with remarkable level of security.

The simplest definition of a homomorphic encryption scheme cryptosystem is a system whose decryption is a morphism, mathematically equivalent with:

$$Let\ \zeta\ be\ a\ mapping\ function\ between\ two\ groups\ (A,\circ)\ and\ (B,\diamond)\ such\ that:$$

$$\zeta\ (x\ \circ\ y) = \zeta(a) \diamond \zeta(b)$$

$$for\ x, y \in A\ \ and\ \ \zeta(a)\ , \zeta(a) \in B$$

Before the major breakthrough of Craig Gentry in 2009 [13] with its ideal lattice-based fully homomorphic encryption, there was a pre-FHE era, where different proposed encryption schemes were presented. Those schemes mainly supported one of the two operations, either addition or multiplication, with a limited number of consecutive operations to be applied on ciphertexts. They are known as schemes with limited unbounded encryption operations. Among them, the most widely adopted were the following:

- *RSA cryptosystem* [10] – with unlimited number of modular multiplications;
- *ElGamal cryptosystem* [11] – with unlimited number of modular multiplications;
- *Paillier cryptosystem* [12] – with unlimited number of modular additions;
- *Goldwasser–Micali cryptosystem* [14] – with unlimited number of exclusive or operations
- *Benaloh cryptosystem* [15] – with unlimited number of modular additions. It was an extension of the *Goldwasser-Micali cryptosystem* which allows larger blocks of data to be encrypted at once;
- Ishai-Paskin cryptosystem [16] – for polynomial-size branching programs. It presented a feasible public-key encryption system appliable to the partial homomorphic encryption schemes;
- Boneh–Goh–Nissim cryptosystem [17] – with unbounded number of addition operations but at most one multiplication)
- Sander-Young-Yung system [18] – with limited number of operations applied to the ciphertext encodings over semigroups based on logarithmic depth circuits.

2.1. Types of homomorphic schemes

There are three types of homomorphic encryptions and the primary difference is the degree of supported operations as well as their frequency. Those are:

- *Partially Homomorphic Encryption (PHE);*
- *Somewhat Homomorphic Encryption (SHE);*
- *Fully Homomorphic Encryption (FHE);*

Partially Homomorphic Encryption allows a handful of mathematical computations to be done on ciphertexts. This means that only one operation, either multiplication or addition could be applied an unlimitedly on encrypted data.

Somewhat Homomorphic Encryption schemes usually support a few operations, usually only one like multiplication operation and addition operation up to a assured complexity and to a limited number of times.

Fully Homomorphic Encryption schemes, while still in development, combines the advantages from both worlds, offering a means to apply both addition and multiplication operations on ciphertexts an unlimited number of times producing valid results. Those try to bridge the gap between data processing and data privacy, by allowing to perform different computations on already encrypted data, thus ensuring privacy.

Being a relatively new domain in the cryptographic world, there were many proposed fully homomorphic encryption schemes that were categorized based by their features and used methods into generations. Each generations tries to build from previous one, adding new optimizations or eliminating some limitations. Currently, there are three generations of homomorphic encryptions, each heaving at least one or more schemes that characterizes and summarizes its generation.

The first generation of fully homomorphic encryptions schemes was represented by Craig Gentry's use of ideal lattice-based cryptography schemes in 2009 [13]. This scheme supports both additions and multiplication on ciphertexts. Although a revigorated and fresh concept in the cryptographic world, the first iterations of various encryptions schemes suffered from performance issues and limited processing operations. The main problem is that every operation that is homomorphically applied to ciphertexts induces a level on noise into the encrypted data. If there are too many operations are applied on the ciphertext, the noise will be greater than a certain threshold of entropy which will make the encrypted data become useless information that could not be decrypted. In order to combat this, ideal lattice-based homomorphic encryptions like Gentry's are heavily reliant on two techniques called *Squashing and bootstrapping.* Despite introducing a large overhead and computation effort, those techniques allow to erase same of the noise induced by homomorphic operations.

Squashing and *bootstrapping* were first introduced by Gentry in 2009 [13], when he proposed "the first fully homomorphic encryption scheme". The main idea behind Gentry's scheme was to create a "Fully Homomorphic Encryption scheme" [13] (FHE) from "a Somewhat Homomorphic Encryption scheme" [19] (SHE), while preserving or reducing the level of entropy. *Squashing* was used to transform SHE to FHE and then to lower the entropy *bootstrapping* was applied.

In simplified terms, according to Rocha and López [20], *squashing* is a procedure in which the encrypted text is compounded by a set of vectors which cut down the degree of the decryption polynomial so that the scheme could handle it. The set of vectors and the security of this process are based on *Sparse Subset Sum Problem (SSSP)* [20].

Bootstrapping is a method in which given a ciphertext *c* with a level of error *e*, will output a ciphertext *d* with an error of $e' << e$. This procedure allows more homomorphic operations to be performed on the ciphertext.

The second generation was focused on improving the performance of the first iteration, developing somewhat and fully homomorphic cryptosystems. The main optimization appeared in the form of a *relinearization* and *modulus switching* techniques that addressed the defects of ideal lattice-based encryption scheme.

Modulus switching was introduced in 2011 by Brakerski and Vaikuntanathan [21]. This technique makes a modulo reduction of ciphertext *c* defined in a ring R_q, to a ring $R_{p,}$ where $p<q$. The advantage of this is that enables the multiplication of two ciphertexts, while preserving the same level of error.

Relinearization is a method introduced by J. Fan and F. Vercauteren in their proposed encryption scheme from 2011 [19] . According to H. Chen [22], *relinearization* is a procedure in which the ciphertext length is reduced, while keeping the underlaying message intact. *Relinearization* technique is similar to *modulus switching*, yielding better performances.

From this generation, worth mentioning are Brakerski/Fan-Vercauteren (BFV) scheme, proposed in 2011 [19] and Brakerski-Gentry-Vaikuntanathan (BGV) [21] scheme, proposed in 2012.

The third generation and latest generation of fully of homomorphic schemes is represented by Gentry-Sahai-Waters scheme (GSW) [23] and Cheon, Kim, Kim and Song (CKKS) [24] scheme.

GSW scheme is based on learning with errors (LWE) problem and for multiplication operations uses a so called '*approximate eigenvector*' method instead of *relinearization* technique which is very time consuming.

Approximate eigenvector tries to decompose the ciphertexts into eigenvectors that approximates the true value, introducing a small margin of error for security reasons. This

technique helps treating homomorphic addition and multiplication as matrix addition and multiplication, making them asymptotically faster. Despite being faster than most encryptions schemes to that date, the main disadvantage was the increased complexity of multiplication compared with the addition.

Cheon, Kim, Kim and Song (CKKS) [24] approximate homomorphic encryption scheme that supports floating point arithmetic with certain precision. The scheme supports an approximate multiplication and addition. A new batching encoding of plaintext and ciphertext based on round learning with error (RLWE) problem is proposed, as well as a so-called *rescaling method* is being introduced. This transformation of messages helps preserving the precision of plaintext after encoding.

Batch encoding allows the input to be encoded into polynomials, providing a good trade-off between efficiency and security as compared to regular computations on vectors.

Rescaling method goal is to reduce the level of error present in the ciphertext, while keeping the further computational cost constant. It is similar with modulus switching method introduced by the BGV scheme.

2.2. Existing implementations and its limitations

After several years from the publications of those encryption schemes, a number of implementations have started to appear, showing each scheme their strengths, as well as their limitations.

HElib [25] is a widely used and important library. HElib implements the Brakerski-Gentry-Vaikuntanathan (BGV) scheme [21] with Smart-Vercauteren compression of the ciphertext technique and other optimizations as well as Approximate Number CKKS scheme. The design and implementation of HElib are documented in [25] and HElib's used algorithms are documented in [26]. HElib proclaims itself as "assembly language for HE", because is implemented by using low-level programming, which takes into account computer components and hardware constraints, without using the instructions or functions of generic programming languages. Since April 2015, it supports bootstrapping [27] and multi-threading. In an important feature, the assessment of AES [5] was homomorphically implemented in an upper layer of the HElib. Unfortunately, in the web context, the use of HElib library is not currently possible due to web limitations. Due to the level low-level implementation of it and the on-going changes of the WebAssembly Standard, the library is not compatible to be used on web.

TFHE or 'Fast Fully Homomorphic Encryption Library over the Torus' is a library [28] that implements a ring-variant of Gentry-Sahai-Waters (GSW) [23] encryption scheme, based on the Learning With Errors (LWE) [29] problem. It is implemented with the help of several programming languages such as C, C++ and Assembly.

Concrete [30] library is a fully homomorphic encryption (FHE) library that implements Zama's variant of TFHE [28]. TFHE is based on Learning with Errors (LWE) [29], a well-studied cryptographic problem. It is written in Rust [31], which helps to reduce the risk of memory leaks. Even though this library had been released in 2022 in an alpha state and have many optimizations under the hood, it is a build primarily in mind with respect to the x86 architecture. It uses some specific processor intrinsics specific to this architecture, of which WebAssembly do not currently support. It also does not yet support Public-key cryptography.

PALISADE [32] is an open-source homomorphic encryption library written in C++ that implements several encryptions schemes such as FHEW, Brakerski-Gentry-Vaikuntanathan (BGV) and Cheon, Kim, Kim and Song (CKKS). Despite its large coverage of homomorphic encryption schemes, the usage is not as simple, the users needing to compile the source code for their platform each time, tweaking several parameters and constant that directly affects the performance and security of the homomorphic operations.

HEAAN or 'Homomorphic Encryption for Arithmetic of Approximate Numbers' is a library written in C and C++ that implements Cheon, Kim, Kim and Song (CKKS) encryption scheme with bootstrapping technique [33] used to reduce ciphertext entropy. It is designed for fixed point arithmetic and supports approximate homomorphic operations between rational numbers.

Cupcake is homomorphic encryption scheme based of the additive version of Fan-Vercauteren scheme [19] written in Rust. It does adhere to the *Homomorphic Encryption Standard* [34], which states that every product that respects and implements this standard provide the ability to make computation on data that is encrypted, offering at least 128-bit security of its operations. Even though this library could be compiled to WebAssembly, it only supports addition, subtraction and multiplication on a very limited range (0 – 255).

2.3. Homomorphic encryption for the web

Even though there are many library implementations of several homomorphic encryption schemes, most of them are not designed for the web. Almost all of the modules are implemented having in mind the x86 computer architecture, using some specific CPU inline functions, called intrinsics for optimization purposes. Luckily, there are some libraries that are redesigned or ported in such a way that they could be used on the web.

Palisade-wasm is a literal WebAssembly port of the PALISADE library [32, 35]. Although is a big step in terms of web accessibility, this port is 2 to 3 times slower than the native C++ implementation, due to the use of 64-bit arithmetic. (64-bit arithmetic is emulated in WASM), with a cumbersome API, that needs constant tweaking.

She-wasm is a two-level homomorphic encryption based on ElGamal encryption scheme [11] written in part with C++, JavaScript and WebAssembly. It offers only 2 operations between

ciphertext: addition and multiplication. From those only the addition can be performed an unlimited number of times, while multiplication could be done only once. It does not yet adhere to the *Homomorphic Encryption Standard* [34], which states that every product that respects and implements this standard provide the ability to make computation on data that is encrypted.

3. Solution architecture

The solution architecture is composed from several parts: a native code library, a WebAssembly WASI compliant transpiler, source code, WebAssembly Compiler, WebAssembly module and a WebAssembly wrapper component.

The native code library is represented by any library that implements at least one homomorphic operation, done on a somewhat or fully homomorphic encryption scheme. This library is preferably implemented in a speed and efficient programming language like C, C++ or Rust that could be further compiled to WebAssembly

The WebAssembly WASI compliant transpiler will convert native library binaries to be able to interact with my source code. This is crucial, because it will give me the possibility to call native functions directly into my written code.

The source code is a part of my implementation written in a language that supports WebAssembly technology.

The WebAssembly compiler is the component that will bundle my code, as well as the native calls into a WebAssembly module with a minimal and rough API written in JavaScript.

The WebAssembly wrapper is represented by my other part of the code, where I will simplify the API and better document, as well as handle some cases of related to the memory usage and garbage collection of memory in the web context.

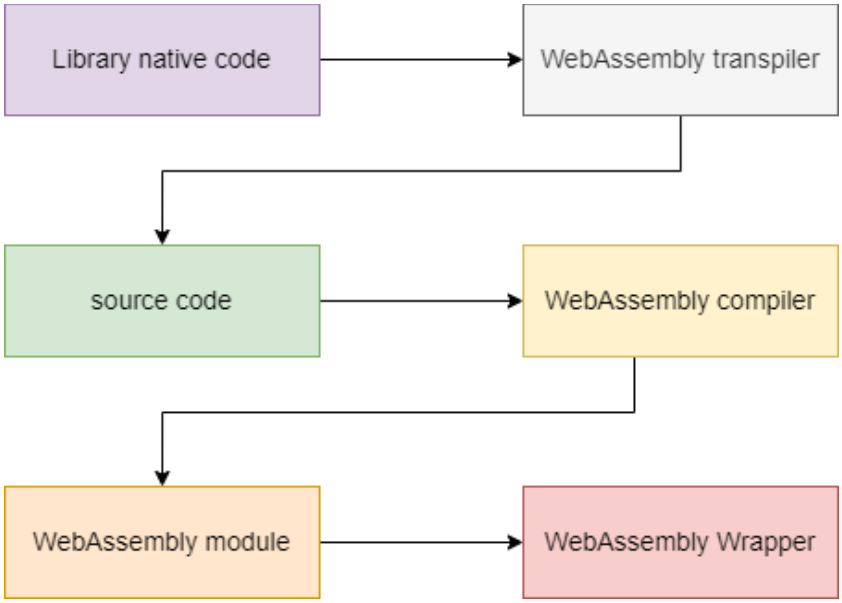

Figure 1. Solution architecture

4. Technology stack

In every project or practical thesis, the technology stack represents an important component in the development of said application. This library uses web-oriented technologies such as WebAssembly, Typescript, Webpack or node-SEAL, as well as performance-oriented technologies such as Rust, Microsoft SEAL in order to efficiently deliver a cohesive module easy to use for the end-user.

4.1. WebAssembly

WebAssembly is a fairy new developed technology described in the *WebAssembly Specification* [36], an open standard developed by W3CCG (Community Group of W3C). This standard defines all subsequent standards and protocols, like WASI (The WebAssembly System Interface) that makes all the WebAssembly technology.

Wasm or WebAssembly is a format of binary instructions for virtual machines that are based on a stack. It is designed to be target whose compilation is portable for all programming languages, allowing distribution on internet for server and clients programs. The designing principles that this language follows are:

- *Fast* – the execution is at near native code speed, making use of all capabilities of supported hardware;
- *Safe* – the code is executed and validated in a sandboxed memory-safe area preventing security breaches or corruption of data;
- *Well-defined* – the code is precisely and fully defining valid applications and their behavior in an easier way to understand;
- *Hardware-independent* – the code can be compiled on modern architectures, mobile devices, desktop or embedded system alike;
- *Language-independent* – it does not favor any particular programming language, object or programming paradigm;
- *Platform-independent* – it can be embedded in modern browsers, integrated in other environments or run as a stand-alone virtual machine;
- *Open* – applications can interoperate with their environments in a universal and simple manner.
- *Portable* – uses broadly supported architectural concepts that are implemented across modern hardware;
- *Parallelizable* – allows compilation, decoding and validation to be split into independent parallel tasks;

- *Streamable* – allows compilation, decoding and validation to start as soon as possible, a priori all the values have been sent;
- *Efficient* – can be compiled, decoded and validated in a single fast pass with either ahead-of-time (AOT) or just-in-time (JIT) compilation;
- *Modular* – applications can be divided into smaller ones that can be used, stored or transmitted independently;
- *Compact* – the format is of binary nature, making the code that is quick to broadcast compared with native code or typical text formats;

Besides designing principles, WebAssembly is structured around those concepts:

- *Values*
- Instructions
- Traps
- Functions
- Tables
- Linear memory
- Modules
- Embedder

Values

WebAssembly has only four basic numeric types. Those are numbers floating-point numbers on 32 and 64-bit, 32-bit integers that also serve as memory addresses and Boolean type. Most of operations of these types are implemented, including operations of full matrices of integers. The integer types, namely unsigned and signed variables can be used interchangeably, because the compiler treats them as being the same. Instead, the interpretation of integers is done via occurred operations or either by using representation in two's complement.

There is also a vector type on 128 bits, which encompasses different sorts of packed data. The representations that are supported are 2 64 bit or 4 of 32 bit floating-point numbers, or different packed integer values, specifically 4 of 32 bit integers, 2 of 64 bit integers. 16 of 8 bit integers or 8 of 16 bit integers.

Finally, variables can have opaque references which are memory addresses that indicates diverse entities. Contrary to other types, visibility or their size is not observable.

Instructions

The computational paradigm of WebAssembly is built upon on a machine that has a stack as a base. The code is comprised of sequences of code directives that are orderly executed. Instructions modifies values with the help of a stack operand which is implicit. There are of two types:

- Simple instructions – perform simple operations on data; They push and pop arguments from the stack operand;
- Control instructions – modifies the flow of control. They are represented by nested inner components such as loops, blocks or conditionals.

Traps

Instructions could produce a trap, which stops the running of the application. Traps are simplified exceptions that cannot be handled by Wasm, but are passed to exterior environment, where they usually can be caught.

Functions

Code is made up from independent procedures. Each function has an array of data as parameters and returns an array of data which are the results. Functions could signal one another, as well as recursively, creating a call stack that is implicit and which is not accessible through a direct manner. Functions could also have local values that are mutable which could be used as registers.

Tables

Tables are vector sets of opaque values of a specific type of item. It enables applications to access the values indirectly with the help of an index operand, which is often dynamic. The untyped procedure ascribed is currently the only available type for this purpose. This is very useful, enabling table indices to emulate procedure pointers.

Linear memory

This is a continuous, modifiable batch of raw bytes. This memory has initially allocated certain size, but the allocation can also grow dynamically. An application can stockpile or load variables from or to memory at any address. Loads that are of type integer can have specified a store size, which is tiny compared with the actual capacity of that type of data.

Modules

A module is an encapsulation that contains linear memories, tables, definitions for functions, as well as immutable or mutable global variables. Definitions can be imported or delivered under one or multiple callsigns. Modules can also initialize the values for their tables or memory segments restricted to a given offset. They also can define a dawn function, which is run by default in an automatic manner.

Embedder

A Wasm implementation is often incorporated into a host environment. The host environment stipulates how and when the modules are loaded, how exports are accessed, as well as how borrowed imports are handled, including the delimitation of hosts. Those are the

main functions of an embedder and any concrete implementation is specific to every software that supports WebAssembly.

In regards with the actual running of a program, Wasm is divided into three semantic phases:

- Decoding
- Validation
- Execution

Decoding

Modules of WebAssembly are bundled in a format, which at its core, it is of binary nature. Decoding is a process in which the WebAssembly bundle is formatted and converted into an internal representation, named abstract syntax that could eventually compiled to machine code directly.

Validation

After being decoded, the module goes through a series of checks regarding the format, safety and meaningfulness of it. In particular, it checks types of the functions, as well as the types of values that are encapsulated into instructions, ensuring that the operand stack is used correctly.

Execution

After being decoded and validated, the code can be executed. Execution is divided in two phases:

- Instantiation– An instance is a dynamic representation of the module, containing its own executing stack and state. Runs the module body, along with all imports and given definitions, initializing global variables, tables, memories that are of linear nature as well as invoking the start function if defined.
- Invocation – Once created, further Wasm module computation can be performed by referencing delivered functions from the module.

The main goal of this technology is to offer near native execution speed to the web, taking advantage of the common capabilities of all current hardware equipment. The safe characteristic is represented by the fact that all the code is verified and executed in a memory-safe, sandboxed environment, which prevents security breaches or data corruption. It is portable, because it is not bound to any language, platform or even hardware. It can easily be compiled from any programming language to any modern architecture without effort.

Due to its versatility and portability, it has helped me to embed compiled native code, which now could be used on the web [37, 38] to implemented the homomorphic encryption library.

4.2. Rust

Rust [31] is a general-purpose programming language, that put emphasis on code and memory security, speed and concurrency. It is often considerate an operating system language due to following features:

- Zero-cost abstraction
- Error messages
- Type inference
- Move semantics
- Data race-free threads
- Pattern matching
- Guaranteed memory safety
- Efficient C bindings
- Safe memory space allocation
- Small runtime
- Cargo

Zero-cost abstraction

Zero-cost abstraction is a higher-level programming concept, like collections or generics that will not add additional runtime cost, only compiler time cost, meaning that the code will be slower to compile, but it will not have any penalty cost when ins run. In Rust, zero-cost abstractions are used to move some behaviors from run time to compile time. Those types do not have any data attached to them, so they do not have any memory representation at the runtime.

Error messages

Error messages are displayed texts directly given by the compiler, when the compilation has failed. Compared to C++, Rust error messages goes one step further in terms of clarity. Error messages are formatted with suggestive colors, the erroneous part is highlighted and often, for the common problems there are meaningful suggestions that can fix the error.

Type inference

Type inference is a feature that increase code readability, as well as the productiveness of software engineers. This feature allows the compiler, in some cases to decide what is the type of certain variables, based on their values, without explicitly specifying it.

Move semantics

For a value-oriented programming language, Rust implementation of move semantics is a big step towards a better optimization and representation of uniqueness invariants. In contrast with C++, which implements move semantics as moves which are not destructive, where variables that were moved from are still available, Rust has chosen to implement the destructive approach. Here, all the variable types can be moved and all operations that involves moving boil down to the copying of the original value to a newer address. Contrary to C++, in Rust we must explicitly call the copy function, thus preventing by default unused or leaked memory.

Data race-free threads

A data race occurs when two or more threads try to access, read or write a resource in the same time. Due to its ownership system, which states that at any given moment a variable can be used at most once. It only transmits the owners of different objects to different threads, resulting in a collision-free running of the program.

Pattern matching

Patterns are a special syntax in Rust for matching different type, simple or complex. They are describing the shape of the variables that are being used. Usage of patterns, alongside match expressions give you more control over program flow. A pattern is a combination of the following:

- Destructured Enums, arrays, tuples or structs;
- Wildcards;
- Literals;
- Variables;
- Placeholders;

Guaranteed memory safety

This is one of the core traits of the Rust programming language. Memory safety is achieved in Rust, by relying on two concepts: Ownership and Borrowing. The compiler implements an affine type system to keep record of ownership of each value: a value can be used at most once and after that the compiler cannot reuses again. The borrowing mechanism allows the compiler to temporarily assign ownership of a value to a variable and that said variable is the sole gateway in which the value can be changed. This mechanism removes the dangling pointers problem often encountered in other languages.

Efficient C bindings

This provides means for Rust to be able to interoperate with C language as it talks to itself. Providing a *foreign function interface* to communicate with C API, as well as leveraging its ownership system to guarantee the memory safety of the programs.

Safe memory space allocation

As many statically typed languages, memory management in Rust is manual, the programmer having explicit control over when and where memory is allocated or deallocated. Compared with C, which uses *malloc* to allocate and then initialize it, Rust fuses those tow operation in one, with the help of the tilde operator (~). This operator returns a smart point to integer that is automatically deallocated when it is unused or out of scope.

Small runtime

This feature is present in all non-assembly programming languages. However not all runtimes are created equal. On the one hand, higher level languages such as Java, C#, Python or JavaScript have big and complex runtimes that are doing automatically memory management, having Garbage Collectors that allocated and deallocated the memory when the program is running. This adds a significant speed penalty to those languages. On the other hand, Rust has most of the code analysis done at compile time as manual memory management. This allows the language to inject a tiny runtime which translated in a significant increase in execution speed.

Cargo

Started as a separated project and later incorporated into the language, Cargo is a build system and package manager. It handles numerous tasks such as building code, downloading dependencies or libraries. *Cargo.toml* is the manifest file that contains metadata such as version, name, URLs to code versioning systems like GitHub, dependencies for packages, which are called 'crates' in Rust.

Compared with C++, Rust has a more aggressive syntax checker, which will prevent the introduction of memory leaks. I chose Rust over C++, because could write code that has no memory leaks and it is fast without optimizing it very much. Being a WASI compliant programming language that could be compiled to WebAssembly, as well as having a more organized library system were also other incentives for using Rust over C or C++.

4.3. Typescript

Typescript [39] is a programming language developed by Microsoft. It is a build upon JavaScript, adding static typing checking to the language. It provides a mature feature system with types and Object-Oriented Programming concepts like classes, interfaces, inheritance etc., expanding other concepts like expressions, statements, modules etc. Besides those, Typescript syntax uses ECMAScript 6 specification [40] or newer and it is able to 'polyfill' older code that could not run-on modern browsers. The polyfill technique is possible due to its core component: Typescript compiler. This compiler takes all the code previously written and it transpiles it to a

JavaScript version of your choice. Most of the time, it targets es3 or es2015, which account for 96% compatibility around all modern browsers. The transpiling step is a process in which all the Typescript code is transformed into JavaScript equivalent and certain features that are newer in the specification are recoded with available syntax, such that it will ensure the desired compatibility will the browsers.

The feature system is comprised of the following elements:

- Types;
- Interfaces;
- Expressions;
- Statements;
- Enums;
- Functions;
- Classes;
- Internal modules;
- External modules and source files

Types

Typescript is adding facultative static types and definitions of said types to the JavaScript programming language. Those types applied on a program components like functions or variables offer a better support and checking during development period. Typescript static assembled type system follow in a close manner the JavaScript run-time system of typing, which is often times a very dynamic, facilitating software engineers to write code in a familiar manner, having their expectations pre-validated by the compiler. The type analysis appears in its entirely at compilation and brings no execution speed penalty to the program.

There are 4 main categories of types:

- Primitive types;
- Object types;
- Type parameters;
- User defined types;

Primitive types are Boolean, Number, String, Void, Undefined and Null along with user defined Enums. The Boolean, number, void, string watchwords are associated with the Boolean, Number, Void and String basic types. The Void type is defined just to pinpoint the vacancy of the value. It is often used as return type to a void function.

Object types are represented by all interfaces, classes, literal and array types. Interface and Class types are made up from interface and implementations of declarations and are called by their given declared name. Interfaces and classes types may also be of generic type, which have one or more type parameters.

Type parameters are actual types that a parameter is bound to a generic function call or a generic type reference. Since those parameters represents various type of parameters have some constraints compared to others. For example, a type argument is not able to be used as an interface.

User defined types are types derived from a primitive Any Type. The any keyword references this. This is the one type has no constraints and can embody any JavaScript value.

Interfaces

Interfaces provide the means of naming and parametrizing object traits, while composing already defined named object traits into new ones. They are a purely compile time construct and do not have any depiction. They are handy for validation and documenting the appropriate form of objects sent as arguments, properties and items returned from procedures. Due to its type system, a Typescript interface type with certain collection of members is treated as indistinguishable, and can be swapped for another interface or type literal with an exact collection of members. Class declarations could include interface references in their definitions causing the need of implementation of said type interface. Despite all advantages, there are some constraints that interfaces have and they must respect them otherwise a compile error occurs:

- An interface cannot directly or indirectly, to be a base type of itself;
- An interface could not state a characteristic with the exact name as a private inherited characteristic;
- Inherited characteristics that have the exact designation must be exact;
- The declared interface of the instance trait must be an imitative type of each of the base ascribed type;
- An interface is allowed to inherit exact piece from various bases types and will contain only one appearance of that particular piece;

Expressions

This feature shows in what manner Typescript handles type checking and type inference of JavaScript expressions. In addition to type checking and type inference, Typescript adds the following constructs:

- Type assertions;
- Super base calls and member passage;
- Arrow function declarations;

Type assertions

A special case of type assertions is the type *this*. Depending on the expression *this* could mean:

- In a member function, constructor, member variable initializer or member accessor, *this* has the class instance type;
- In a static accessor or static procedure, *this* has the type analogous to the type of assembler procedure, without any build up on signatures;
- In a standard function expression or in a procedure acknowledgment, *this* has the type of Any;
- In a global module, *this* has the type of Any;

Another assertion type is when the expressions are used as *identifiers.* When this occurs, the definition refers to the class that is the most nested, module, Enum, variables, function or argument which points to the location. The type of definition is matched with the ascribed type entity:

- For a class, the assembler type matched with the assembler procedure object.
- For a module, the object type is the module instance type.
- For an Enum, the object type is the Enum object type.
- For a variable, the type is the variable type.
- For a function, the task type is the procedure item type.
- For a parameter, the type is the parameter type.

Super base calls and member passage

Super calls are made from the keyword super followed by parameter array confined in parentheses. Those types of signals are allowed only in assemblers of imitative classes. A super signal calls the base assembler on the instance ascribed by this and it is done as a procedure signal. The super call expression type is Void. Module or class member access is done by using the dot operator. Originally, JavaScript did not have access modifiers, but recent ECMAScript proposals include two types of modifiers: public and private. The public modifier marks the said property or function to be accessible to everyone that has access to the object that contains those members. The private property modifier is explicitly saying that the property is available only in the class instance. Those features were also implemented in Typescript, much earlier in a unusual way: the private variables would contain at the beginning of their names the hashtag symbol (#).

Arrow function declarations

Arrow function or lambda function are a concise form of a normal Typescript/ JavaScript function, with some differences:

- Lambda functions do not have their own bindings to arguments, super or this and should not be used as methods.
- Lambda functions do not have access to the *new target* keyword
- Lambda functions cannot be uses as constructors

- Lambda functions cannot use yield, in their function bodies.

I used Typescript in this library because I wanted for the end-user to have the same amount of control over the variables and avoid common memory leaks. The application is composed from several parts, which insists on the improvements brought by the object-oriented programming [20] such as code reutilization, malleability, separation of concerns and modularity.

Statements

A statement is syntactic construct of a programming language that expresses the need of some action to be carried out. Those can be simple statements like assign, call or return or complex or compound like do while, while do, switch or if-else statements. In this regard, Typescript enhances those statements, by providing pre-validation, type inference and type checking which eliminates common errors like unreachable code, improving intersection and union of types.

Enums

Enums are a distinct subtype of the primitive Number type that has a set of constants associated with a set of values of various types. The Enum body is an unnamed object variable that contains a set of characteristics, all of Enum trait, matching the values defined in the Enum body declaration. Furthermore, Enum object type include a numeric index, that make possible to easily iterate through the Enum representative. Enum representative are the named values of the Enum type. Those representatives can be categorized in two:

- constant representative
- computed representative

Constant representatives have constant values known prior to the compilation, allowing them to be substituted in place of references. Computed representatives have values that are calculated at run time and are not known at compilation, denying them the possibility to be used as substitution variables.

Functions

Typescript extends JavaScript functions by including return type annotations, type parameters, parameters, overloads, res parameters and default parameters values. Rest parameter syntax allows Typescript and, implicitly JavaScript to allow a function to have an infinite number of arguments, passed as an array. Default function parameter let the named parameters to have a default value if that argument is not passed on the function call occurs.

Classes

Typescript implementation of classes is closely aligned with proposed ECMAScript standard [40], including expansions for static representative definitions, instance and characteristics created and initialized from assembler arguments. Class declaration introduces

named traits and give concrete implementations for them. Classes can inherit other classes or implementing other interfaces, allowing imitative classes to expand their behavior by extending base classes. The definition of a class must include the declaration of a class type and an assembler procedure, both having the callsign given by identifier. The class type is made from the defined representative instances from the class object and the representatives which come from the base class. The assembler procedure is made up from the constructor declaration, the static representatives inherited from base class and the static representative declarations from the class body. The assembler setups and returns an item of class type. Other properties and feature may include:

- A class can also have trait arguments, that are placeholders for actual traits that will be supplied when the class is ascribed in trait references;
- A class that has trait arguments is named as a generic class;
- Class that omits the declaration of a assembler, an automatic assembler is provided; In a normal class, that do not extend no base class, the automatic assembler has no arguments and will not perform any action; In a derived class that extends a base class, the automatic assembler has the exact argument list and calls the base class constructor;

Internal modules

A named container of declarations and statements is called an internal module. An internal module encompasses a singleton module as well as a namespace instance. The namespace has the named traits and other namespaces, while the singleton module item has proprieties of its exported members. Internal modules could be of two types:

- Deployed
- Non-deployed

A deployed module is an internal module for which an object instance is created, containing callable functions and usable properties. A non-deployed module is an internal module that only contains only interface types and other non-deployed modules. Visually, a deployed module is one that occupies memory and has concrete implementations, whereas the non-deployed one has no memory representation at run time. The utility of the non-deployed module is mainly for documentation purposes. Importing of a module is an important feature of Typescript. It is used to make local aliases for items in other modules, thus facilitating type reusability.

External modules and source files

Typescript implements external modules using the ECMAScript standard [40] as their guideline. It also supports code generation targeting different module systems such as CommonJS [41] and ECMAScript modules or AMD modules [42] for both web and Node.js [43] applications.

A Typescript program is made up from one or more source documents that are either declaration sources documents or implementation source documents. Source documents with the ".ts" addition are implementation source documents and they are containing statements, declaration and implementations. Declaration source documents have as addition ".d.ts" and they contain only definitions. Those documents are a strict subset of the implementation documents. When a program is amassed, all of the program's source documents are bundled up and prepared together. Some definitions and statements from distinct source documents could depend on each other, in a circular way. By default, a JavaScript output document is created for each source document implementation, but no output is made for the declaration documents.

Definition files are restricted to contain only definitions. Those documents are used to define stationary trait information for current JavaScript codebase in complementary manner. They are facultative, but they allow the Typescript compiler and its subsequent apparatus to give better support and checking when integrating a Typescript application with an existing JavaScript codebase.

Implementation and definition of source documents that do not have any export or import definition is the "single *global module".* Entities stated in the global module are visible all over the application. The order of initialization of the source documents that constitutes the global module banks on the import order of the JavaScript generated files, which are loaded at run time. A way in which import order could be controlled is by the <script/> tags that references those generated JavaScript files.

Declaration and implementation source files that contain at least one export or import declaration are treated as distinct *external modules*. Entities defined here, are visible only in module scope, but the exported entities can be visible to other modules via import declarations.

External modules are distinct chunks of code ascribed using external modules names. They are written as separate files that contain at least one export or import declaration. If the export declaration contains the *export* keyword, the source document is treated as an external module; otherwise, the document is treated to be part of the global module. In order to export those modules, one must follow these naming conventions:

- An external module callsign is a sequence of characters, fenced by forward slashes;
- The series of characters must be a camelCase identifier, ".." or ".";
- External module callsigns will not have document callsign additions like ".js";
- External modules may be "top-level" or "relative". An external module callsign is "relative" if the first characters are ".." or "."

4.4. Webpack

Webpack [44] is a module bundler that makes a library or application to be compatible with modern browsers and Node.js [43] applications. The whole processing is based on an internal *dependency graph* created from one or more *entry points*, that will connect every module of the application into one or more *bundles,* which are static assets that can serve the content from.

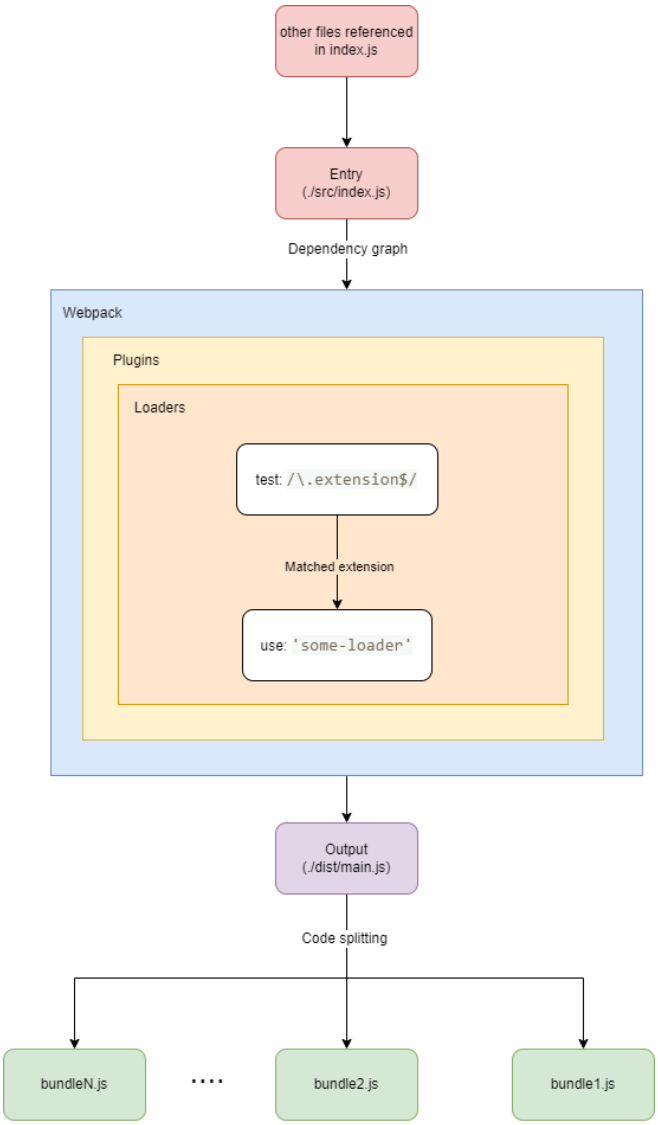

Figure 2. Webpack bundling process

As is shown in figure 2, the Webpack processing is based on the following core concepts:

- Entry
- Dependency graph
- Loaders

- Plugins
- Output
- Other features

Entry

An entry point tells to the Webpack, which is the first file that it needs analyzing in order to create the internal dependency graph. Based on this, the program will figure out directly and indirectly, which other libraries and modules depends on the entry file. By default, the entry file is in *./src/index.js*, but can be changed via a webpack configuration.

Dependency graph

Whenever a file is dependent on another file, Webpack treats this as a dependency. By doing so, it creates the dependency graph which allows the program to contain also non code assets, such as web fonts and images and makes the bundling process customizable, in the sense that the program can create a single monolithic bundle or smaller ones depending on the Webpack configuration.

Loaders

Loaders are small toolchain program that transforms the source code of a module. They allow to process files before using it in the application. Loaders have two configurable properties: the *test* property and the *use* property. The *test* property, which is often a Regex expression, match the files that will be transformed. The *use* property tells Webpack which loader should be used in the transformation process.

Plugins

Plugins are the spine of Webpack. They provide a better and a wider range of tasks than loaders. If the loaders are mainly concerned with file transformation, plugins deal also with asset management, bundle optimization and injection of environment variables. Plugins are JavaScript object that contain an apply method. This method is called by the Webpack compiler, having access to the entire compilation cycle.

Output

Output configuration options tells the program how and where to write the compiled files. It is worth mentioning that there could be more entry points, but only one configured output.

Other features

Webpack offers out of the box three main modes: development, production or none. They enable Webpack built-in code optimization to be applied on the compiled bundles. Browser

compatibility is also another aspect that Webpack gracefully handles, supporting all browsers that are ES5-compliant. Another great feature is the HMR or Hot Module Replacement. This add, removes or exchanges modules while the program is running, without a full reload. HMR can significantly speed up the development process because it retains the application state, which is normally lost when a full reload occurs, swaps out only changed modules and instantly updates all the modifications made in JS or CSS [45] code.

In my case Webpack is crucial because on one hand, it generates a JavaScript bundle of the WebAssembly module that can be imported in the Typescript library. On the other hand, it is used to generate CommonJS [41] and ECMAScript modules [42] for both web and Node.js applications.

4.5. Microsoft SEAL

SEAL or 'Simple Encryption Arithmetic Library' [46] is a Microsoft developed homomorphic encryption library written in C++ that implements several encryptions schemes such as Brakerski/Fan-Vercauteren (BFV), Brakerski-Gentry-Vaikuntanathan (BGV) and Cheon, Kim, Kim and Song (CKKS). SEAL is part of the native code component of the module, due to its stability and features in comparison with other homomorphic encryption libraries. [47, 48]

4.6. node-SEAL

Node-seal [49] is literal JavaScript/WebAssembly port of the Microsoft SEAL library. Although is a step towards running homomorphic operation directly in the web, it has a memory management problem. Due to its implementation, all the objects created involved in the homomorphic operations must be manually deallocated, thus not taking advantage of the JavaScript's Garbage Collector.

I choose this module to be part of the implementation because it speeds up the compilation process. Instead of manually compiling the whole SEAL library that was limited by the targeted CPU architecture, node-seal offers already access to the WebAssembly compiled, which reduce the development process by a significant margin.

5. Solution implementation

After exposing a general architecture in the chapter 3, the chapter 5 will contain the concrete implementation of the library, made with technologies thoroughly presented in the previous chapter, all the subprocesses that conducted to its creation, as well as the advantages that arise from using the EasyFHE library. The creation subprocesses will be detailed in the following sub-sections. The module is divided in five distinctive components, as shown in the below figure:

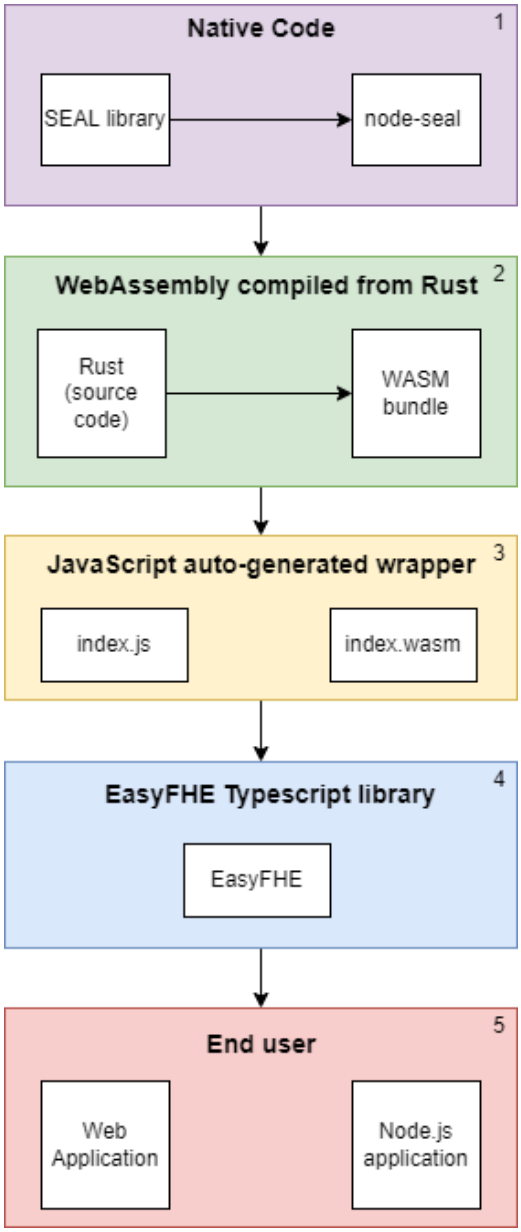

Figure 3.EasyFHE component overview

5.1. Library compilation process

The first subprocess is represented by the library compilation process, which is made up from the first three components of EasyFHE, namely "Native Code" component, "WebAssembly compiled from Rust" component and the "JavaScript auto-generated wrapper" component. For a clearer view, below is presented a more detailed scheme of the process.

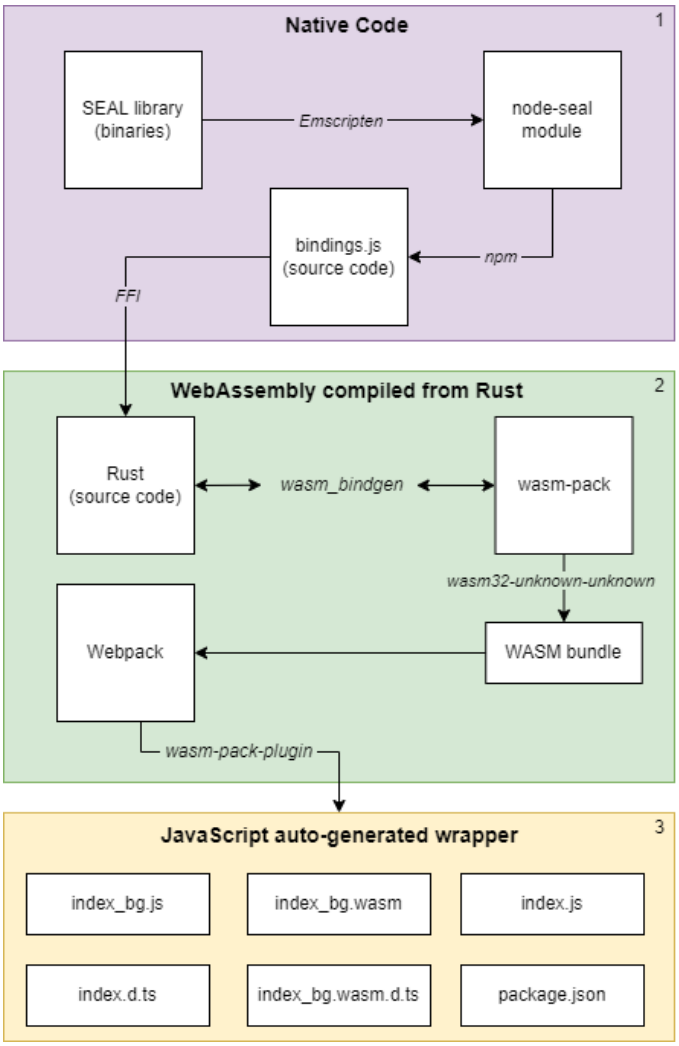

Figure 4.Library compilation process

Firstly, the "*Native Code*" component is comprised of the SEAL library binaries, node-seal module and a bindings file written in JavaScript. The SEAL library is compiled to a WebAssembly bundle using *Emscripten* [50] and after it is embedded into the node-seal module. The *bindings.js*

file consists of a much more simplified SEAL API, as well as a better memory management implementation that will be used in the Rust context via *FFI* [51]. FFI or Foreign Function Interface allows the execution of external code from other languages in the Rust environment.

As an implementation problem, before SEAL and node-SEAL libraries were considered to be used, I tried other libraries that could would not need the bindings.js file, but they proved to be very limited in their capabilities. First attempt was an implementation using concrete library. Concrete [30] library is a fully homomorphic encryption (FHE) library that implements Zama's variant of TFHE [28]. TFHE is based on Learning with Errors (LWE) [29], a well-studied cryptographic problem. Despite being written in Rust [31] which would have made my code to be easier to interact with it, the library is built with the x86 architecture in mind. It uses some specific processor intrinsics specific to this architecture, of which WebAssembly do not currently support. It also does not yet support Public-key cryptography which is a key feature my module. Second attempt was using the cupcake library. Cupcake is homomorphic encryption scheme based of the additive version of Fan-Vercauteren scheme [19] written in Rust. This implementation was rapidly abandoned when I discover the fact that the library will only perform reliably with the values in range of 0-255. Apart from this, there were some other lacking features like plain or cipher subtraction, exponentiation and negation of a cipher. The lack of documentation was another factor that tipped the scale in the favor of SEAL and node-SEAL libraries.

The second component is represented by the Rust source code, Webpack [52], *wasm-pack* program and its output, WASM bundle. The Rust source code consists of exportable functions that can run via *FFI* the functions defined in *bindings.js*. Those functions are decorated with the *#[wasm_bindgen] macro* attribute [53] that will automatically create mirror representation of the types and values from the Rust environment to the WebAssembly one. This macro generates the glue code that will be able to run in any WASI compliant programming language. Here, it allows code written in Rust to be executed in the JavaScript and vice-versa. *wasm-pack* is a Rust toolchain that will simply to the compiling and budling to WebAssembly given a certain build target.

Here, the build target is *wasm32-unknown-unknown* which tells to the *wasm-pack* to compile all the code with respect to the standard WebAssembly on the 32 bits. Due to the infancy of the WebAssembly environment, the 32-bit architecture is the only stable target build, 64-bit variant being emulated. After the program is finishing its running, it produces a WebAssembly Module bundle. (*WASM bundle*). Having the WASM bundle as an input, the Webpack program with the help of a plugin called *wasm-pack-plugin* will generate the third component of the library, namely the auto-generated JavaScript wrapper.

The third component is a lightweight wrapper for the WASM file that will provide an easier means to use, import and export in other projects as well as a good compatibility coverage between node applications and modern browsers.

5.2. Typescript module creation process

The Typescript creation process is represented by the last two components of the EasyFHE library. The first component is the Typescript library itself and the second component is the what the end user will ultimately use in his projects.

The first component is made up from Typescript, Webpack and the EasyFHE codebase. In order to have a clear picture of the code, as well as to understand the library capabilities, I developed a simplified scheme, that it is shown in the following picture:

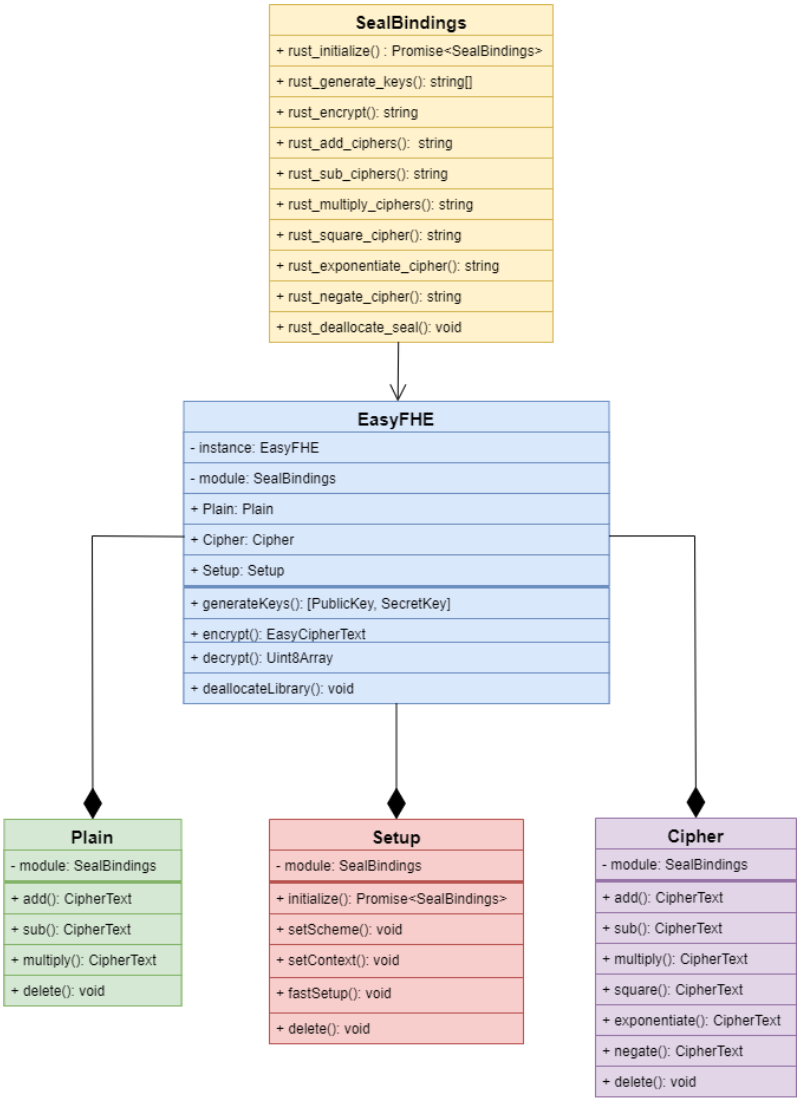

Figure 5.EasyFHE simplified representation

The simplified internal structure of the EasyFHE library is comprised of the following classes:

- SealBindings – a bridge between SEAL-Rust code and Typescript
- EasyFHE – main endpoint for the developers to interact with it
- Plain – contains methods specialized in operations with plaintexts
- Setup – deals with the initialization of homomorphic encryption schemes
- Cipher – contains methods specialized in operations with ciphertexts.

SealBindings class represents the code made in Rust, which further communicates with SEAL and node-SEAL libraries, containing some helper methods which simplify the native calls. *EasyFHE* class is the main point of interaction between the developer and library. It is implemented using a combination between a Façade design pattern and a Singleton design pattern. In order to have a granular control, *EasyFHE* class contains three instances of *Plain*, *Setup* and *Cipher* class, each class implementing specific methods for their respective use.

The Typescript module creation process, comprised of the EasyFHE library component and the End user component, is illustrated in the following picture:

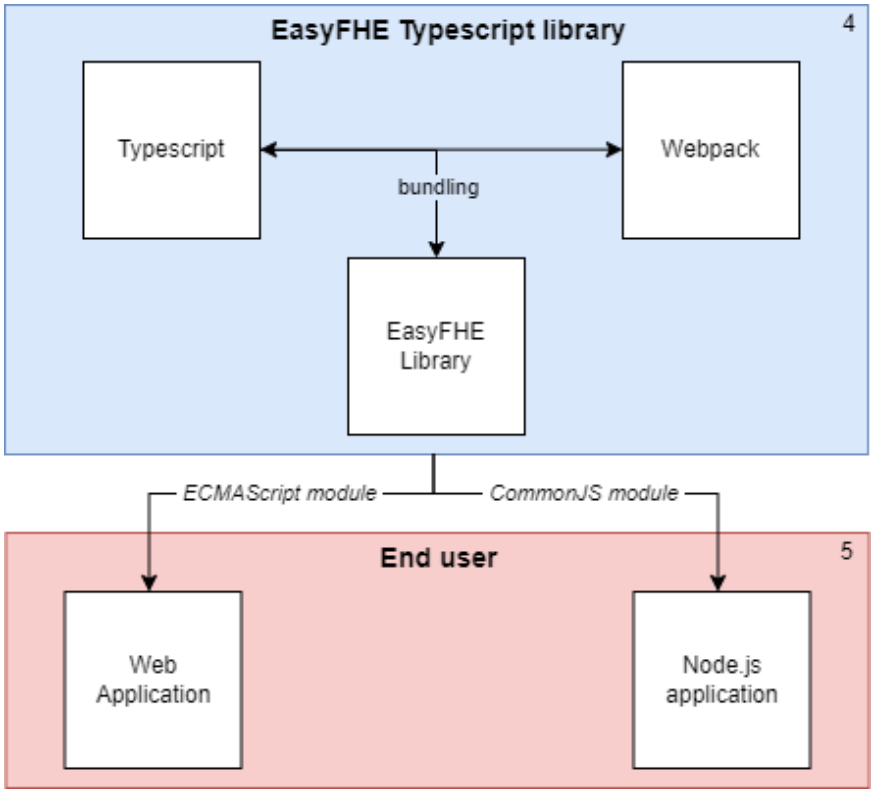

Figure 6.EasyFHE and its usage

Firstly, the Webpack allows the JavaScript auto-generated wrapper to be imported into the Typescript library. Here, there are defined types, alongside functions with comments and explanations, as well as a further simplification of the API that is better integrated with the most modern IDEs.

After the Typescript compiles the library and generates the JavaScript equivalent code, Webpack will take that as an input and it will create 2 separate versions of the final library: an ECMAScript module good for any web application and a *CommonJS* one, fitted for a Node.js application.

5.3. Ease of use

The EasyFHE library is characterized by its capability to be used by almost everyone, without having strong concepts regarding homomorphic encryption schemes. Adding to this argument, a demo web application was developed.

The web application was developed with Vue.js [54] and Quasar [55] frameworks. Vue is a versatile, performant open-source model-view-view-model written in Typescript which is used for building user interfaces and single page web application. The building block of the framework it is represented by the Vue components.

Vue components extend HTML elements, encapsulating reusable code. Semantically, each component it is made up from 3 elements: template, state and style. The template is where the HTML structure is defined and decorated with function as event listeners. The state is a JavaScript object that contains all the business logic, behavior and state associated with the component. Finally, the style part is represented by the CSS [45] or CSS-derived syntax, like SCSS or LESS, used for improving the template design.

Among other features, it possesses a very mature reactivity system focused on speed, which allows the data to be in sync with the displayed HTML [56] elements. In order to achieve its advertised speed, Vue uses a virtual-DOM (document object model), which permits to all components to be reactive, thus sending minimal changes to the actual DOM.

Based on Vue, Quasar framework is open-source project for building application based on Vue.js. It contains a highly-customizable collection of commonly used components used in web development. Production focused, Quasar offers out of the box custom Vue components, mobile friendly and responsive, ready to be deployed on Web as PWA (Progressive Web Application), SPA (Single Page Application), SSR (Server-Side Rendering), as a Mobile Application, using Cordova for Android and iOS and as a Desktop application, using Electron for Windows, Mac or Linux, all under a single codebase.

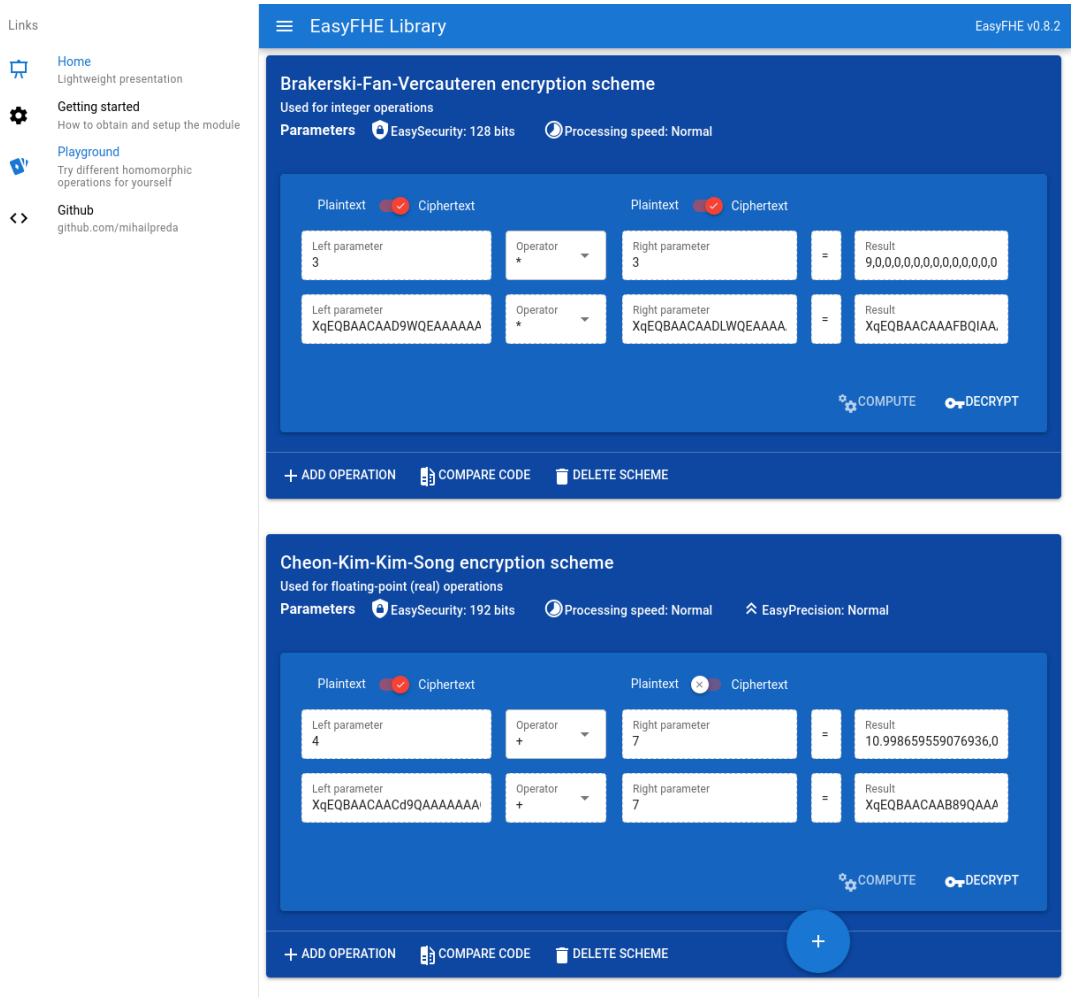

Figure 7. Demo UI

As it is show in figure 7, the web application has four pages:

- Home – It contains a lightweight presentation of the library;
- Getting started – Explains how to setup and use the library;
- Playground – It is a page that contains a demonstration of library usage; It has a friendly UI, supporting BFV, BGV and CKKS encryption schemes, as well as a part of the homomorphic encryption operations, namely: addition, subtraction and multiplication; Each operation supports two computation pairs: plaintext – ciphertext and ciphertext – ciphertext;
- GitHub – It contains an external link to my GitHub profile, were is all the code regarding this library.

In order to see the efficiency and features of the library in comparison with node-SEAL counterpart, a scenario was developed. The ACME company has more than 1000 employees and want to establish the budget for the next year. For this to happened, the company chief financial officer wants to compute several metrics:

1. It will add up the number of hours from the past 4 quarters for each employee.
2. It will compute the current expense meaning the salary of each employee by multiplying hourly wages with the worked hours form the 4 quarters.
3. In order to have a forecasted profit, the CFO will compute a year-to-year expanse indicator, by subtracting the current expense from the last year expense.

The computation of the first metric can be assimilated to a addition of vectors and it will use Brakerski/Fan-Vercauteren (BFV) scheme. The second one translates to an multiplication of vectors and will use Cheon, Kim, Kim and Song (CKKS) approximate homomorphic encryption scheme. Lastly, the third one is equivalent with a subtraction of vectors and will use Brakerski-Gentry-Vaikuntanathan (BGV) encryption scheme. Having privacy in mind, all of the metrics will be computed having a level of security of at least 128 bits. After doing all the computations, the following results have been observed:

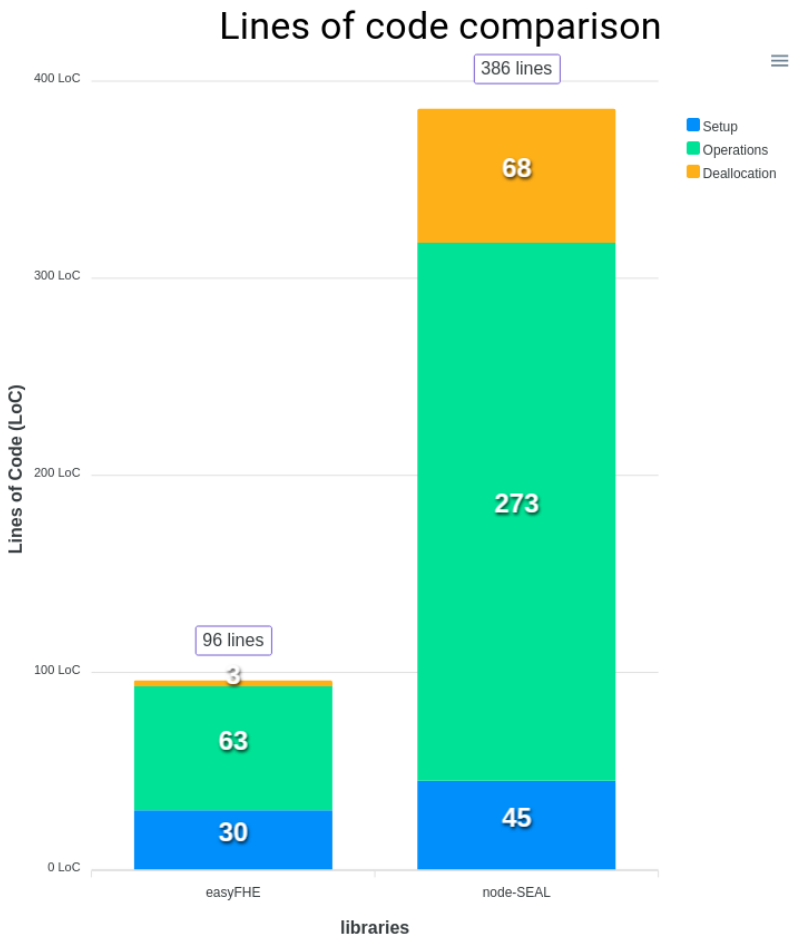

Figure 8.Lines of code comparison

As show in figure 8, the EasyFHE source code is on average 24.87% from the node-SEAL equivalent or almost 4 times more compact, being much more developer and production friendly. Memory management is another key feature of the module reducing the overall number of objects to be manually deallocated by the user, as well as reducing the memory footprint.

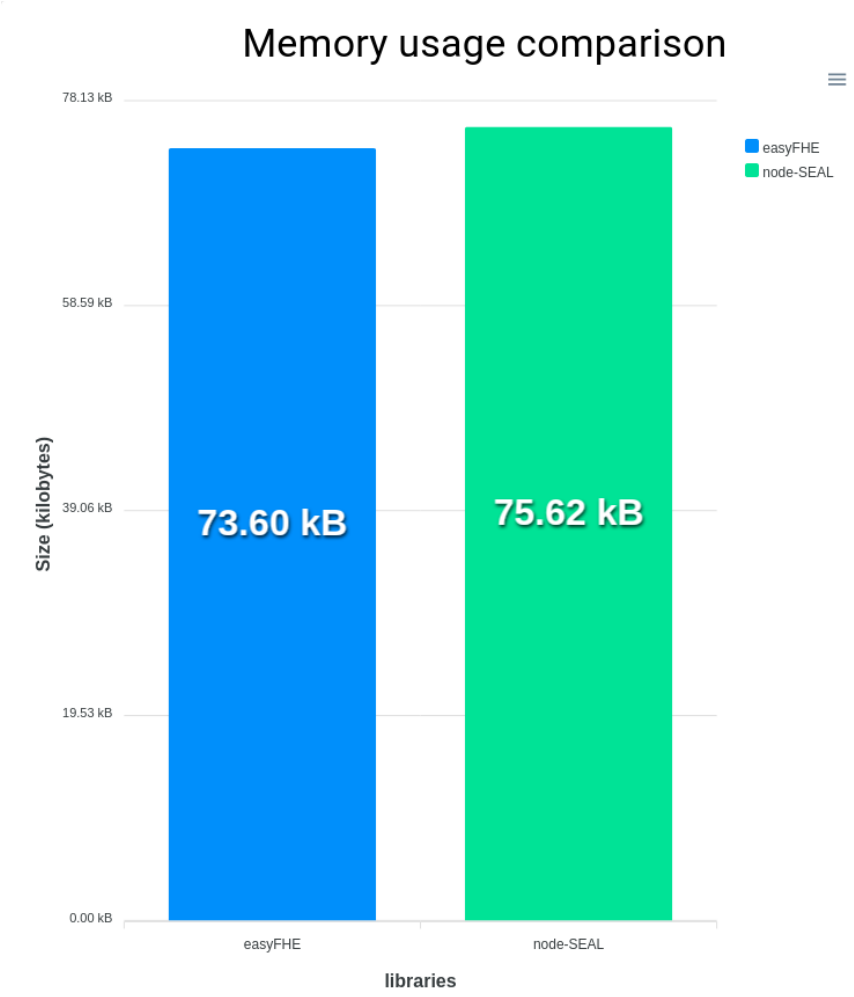

Figure 9.Memory footprint

As is shown in figure 9, the module uses on average with 2.67% less RAM memory due to the optimization performed in bindings.js file, as well as in the Rust code. While developer friendly code and optimized memory are important things to consider when using this library, the speed factor cannot be ignored.

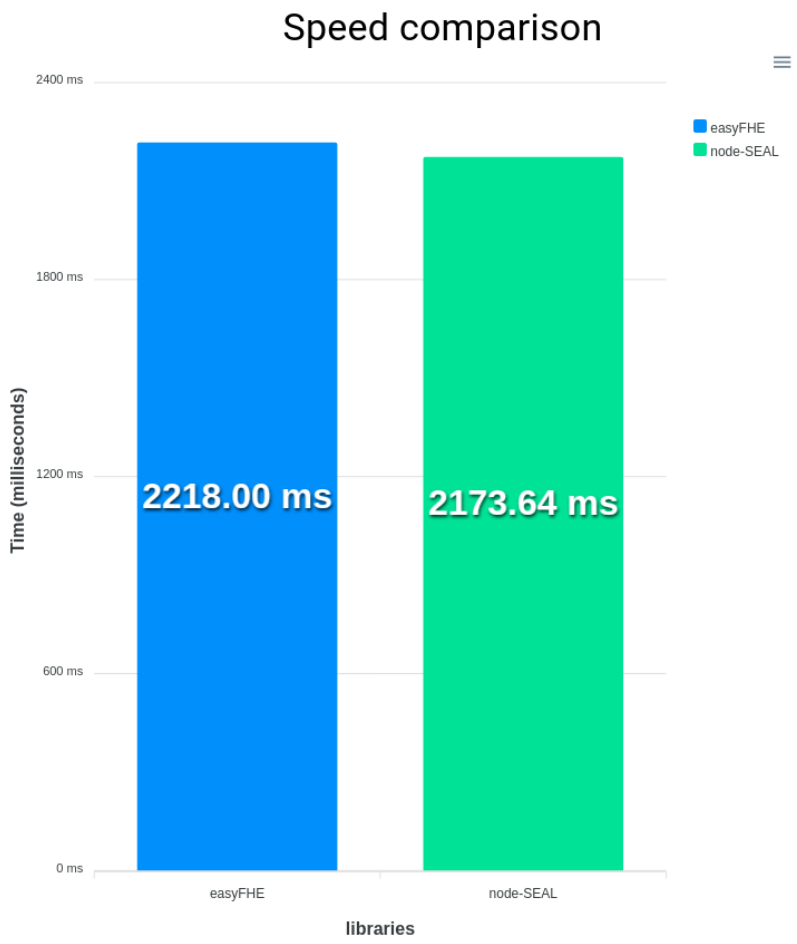

Figure 10.Speed comparison

We can see in figure 10 that the EasyFHE is on average with 2.04% slower than the node-SEAL counterpart. This increase in processing speed is due to the added WebAssembly and Typescript overhead, as well as from the Webpack's experimental loading of asynchronous WebAssembly modules.

6. Conclusions

This thesis tried and succeeded to make homomorphic encryption more accessible to the web, as well as easier for developers to experiment with it.

The module implementation has shown that it has achieved a more compact and less verbose code, with a better memory management, at the expense of a 4 to 5 percent of the execution speed. Along the way, it has demonstrated how different languages can interact cohesively through WebAssembly technology.

Although a revolutionary technology, the limitations of WebAssembly make it difficult to have cost-efficient implementations in terms of processing speeds and memory consumption.

Future version of the module could be based on a GPU implementation that would harness the advantages of SIMD paradigm, decreasing the time cost and increasing the usability of fully homomorphic encryption technology in real world, speed-focused applications.

Bibliography

[1] World Economic Forum, "The Global Competitiveness Report," 2018.

[2] J. Nolin, "Data as oil, infrastructure or asset? Three metaphors of data as economic value," *Journal of Information Communication and Ethics in Society,* November 2019.

[3] M. Loi and P.-O. Dehaye, "If Data Is The New Oil, When Is The Extraction of Value From Data Unjust?," *Philosophy and Public Issues – Tyranny, Democracy, and Economy,* January 2017.

[4] T. B. Patil, G. K. Patnaik and A. T. Bhole, "Big data privacy using fully homomorphic non-deterministic encryption," in *7th IEEE international advanced computing conference, IACC 2017*, Hyderabad, 2017.

[5] F. P. Miller, A. F. Vandome and J. McBrewster', "Advanced Encryption Standard," 2009.

[6] M. Ogburn, C. Turner and P. Dahal, "Homomorphic Encryption," in *Complex Adaptive Systems, Conference Organized by Misouri University of Science and Technology*, Baltimore, 2013.

[7] M. Zhao and Y. Geng, "Homomorphic Encryption Technology for Cloud Computing," in *Proceedings of the 9th International Conference of Information and Communication Technology [ICICT-2019] Nanning, Guangxi, China January 11-13, 2019*, 2019.

[8] X. Yan, Q. Wu and Y. Sun, "A Homomorphic Encryption and Privacy Protection Method Based on Blockchain and Edge Computing," 2020.

[9] S. Mittal and K. Ramkumar, "Research perspectives on fully homomorphic encryption models for cloud sector," *Journal of Computer Security,* vol. 29, pp. 135-160, 29 March 2021.

[10] R. L. Rivest, L. Adleman and M. L. Dertouzos, "On Data Banks and Privacy Homomorphisms," in *Foundations of secure computation*, New York, 1978.

[11] T. Elgamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," in *Advances in Cryptology—CRYPTO'84*, 1985.

[12] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Advances in Cryptology—EUROCRYPT'99*, Berlin, 1999.

[13] G. Craig, "Computing arbitrary functions of encrypted data," in *Communications of the ACM*, 2010.

[14] S. Goldwasser and S. Micali, "Probabilistic encryption," in *Journal of Computer and System Sciences*, 1984.

[15] J. D. C. Benaloh, "Verifiable Secret-Ballot Elections," 1987.

[16] Y. Ishai and A. Paskin, "Evaluating Branching Programs on Encrypted Data," in *Theory of Cryptography Conference*, Berlin, 2007.

[17] D. Boneh, E.-J. Goh and K. Nissim, "Evaluating 2-DNF Formulas on Ciphertexts," in *Theory of Cryptography Conference*, Springer, 2005.

[18] T. Sander, A. Young and M. Yung, "Non-Interactive CryptoComputing For NC1," in *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, Washington, 1999.

[19] J. Fan and F. Vercauteren, "Somewhat Practical Fully Homomorphic Encryption," in *Cryptology ePrint Archive: Report 2012/144*, 2012.

[20] V. F. Rocha and J. López, "An Overview on Homomorphic Encryption Algorithms," 2019.

[21] Z. Brakerski, C. Gentry and V. Vaikuntanathan, "Fully Homomorphic Encryption without Bootstrapping," in *ITCS '12: Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, New York, 2012.

[22] H. Chen, "Optimizing relinearization in circuits for homomorphic encryption," *ArXiv,* October 2017.

[23] C. Gentry, A. Sahai and B. Waters, "Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based," in *CRYPTO 2013: Advances in Cryptology – CRYPTO 2013*, 2013.

[24] J. H. Cheon, A. Kim, M. Kim and Y. Song, "Homomorphic Encryption for Arithmetic of Approximate Numbers," in *Advances in Cryptology – ASIACRYPT 2017*, 2017.

[25] S. Halevi and V. Shoup, "Design and implementation of HElib:a homomorphic encryption library," IBM Research (Manuscript), 2013.

[26] S. Halevi and V. Shoup, "Algorithms in HElib - An Implementation of homomorphic encryption," in *International Cryptology Conference*, 2014.

[27] S. Halevi and V. Shoup, "Bootstrapping for HElib," in *EUROCRYPT 2015: Advances in Cryptology -- EUROCRYPT 2015*, Berlin, 2015.

[28] I. Chillotti, N. Gama, M. Georgieva and M. Izabachène, "TFHE: Fast Fully Homomorphic Encryption Over the Torus," in *Journal of Cryptology*, 2019.

[29] L. Fucai, W. Fuqun, W. Kunpeng and C. Kefei, "Fully homomorphic encryption based on the ring learning with rounding problem," *IET Information Security,* vol. 13, pp. 639-648, November 2019.

[30] Zama.ai, "Concrete - Fully Homomorhpic Encryption Library written in Rust," 2022. [Online]. Available: https://docs.zama.ai/concrete/lib/user/index.html.

[31] S. Klabnik, The Rust Programming Language, No Starch Press, 2018, p. 552.

[32] Y. Polyakov, K. Rohlof and G. W. Ryan, "PALISADE Lattice Cryptography Library User Manual," 29 April 2022. [Online]. Available: https://gitlab.com/palisade/palisade-release/blob/master/doc/palisade_manual.pdf. [Accessed 7 May 2022].

[33] J. H. Cheon, H. Kyoohyung, A. Kim, M. Kim and Y. Song, "Bootstrapping for Approximate Homomorphic Encryption," in *Advances in Cryptology – EUROCRYPT 2018*, Seoul, 2018.

[34] M. Albrecht, M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, S. Halevi, J. Hoffstein, K. Laine, K. Lauter, S. Lokam, D. Micciancio, D. Moody, T. Morrison, A. Sahai and V. Vaikuntanathan, "Homomorphic Encryption Standard," *Academic Consortium to Advance Secure Computation,* 2019.

[35] V. Kadykov, A. Levina and A. Voznesensky, "Homomorphic Encryption within Lattice-Based Encryption System," in *14th International Symposium "Intelligent Systems"*, 2021.

[36] W. C. Group, "WebAssembly Specification," 16 April 2022. [Online]. Available: https://webassembly.github.io/spec/core/_download/WebAssembly.pdf. [Accessed 16 April 2022].

[37] A. Haas, A. Rossberg, D. L. Schuf, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai and J. Bastien, "Bringing the Web up to Speed with WebAssembly," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017.

[38] S. M. Jain, WebAssembly for Cloud, 1st ed., Bangalore: Apress, Berkeley, CA, 2022, p. 163.

[39] Microsoft, "TypeScript: JavaScript With Syntax For Types," [Online]. Available: https://www.typescriptlang.org/. [Accessed 5 May 2022].

[40] ECMAScript, "ECMA-262, 11th edition, June 2020: ECMA Interrnational," 16 April 2022. [Online]. Available: https://262.ecma-international.org/11.0/. [Accessed 2022 April 2022].

[41] N. Bevacqua, JavaScript Application Design, Manning, 2015.

[42] D. Grover and H. P. Kunduru, ES6 for Humans, Apress, 2017.

[43] R. Dahl, "Node.js," 2009.

[44] J. Vepsäläinen, SurviveJS - Webpack 5: From apprentice to master, Independently published , 2020, p. 337 .

[45] World Wide Web Consortium, "CSS Snapshot 2021," 31 December 2021. [Online]. Available: https://www.w3.org/TR/css-2021/. [Accessed 16 April 2022].

[46] H. Chen, K. Laine and R. Player, "Simple Encrypted Arithmetic Library - SEAL v2.1," in *International Conference on Financial Cryptography and Data Security*, 2017.

[47] S. S. Sathya, P. Vepakomma, R. Raskar, R. Ramachandra and S. Bhattacharya, "A Review of Homomorphic Encryption Libraries for Secure Computation," December 2018.

[48] S. M. Fawaz, N. Belal, A. ElRefaey and M. W. Fakhr, "A Comparative Study of Homomorphic Encryption Schemes Using Microsoft SEAL," *Journal of Physics: Conference Series,* vol. 2128, 12 October 2021.

[49] N. Angelou, "node-seal," Morfix.io, 16 March 2019. [Online]. Available: https://docs.morfix.io/. [Accessed 16 March 2022].

[50] G. Gallant, WebAssembly in Action: With examples using C++ and Emscripten, Manning, 2019, p. 425.

[51] T. Rustonomicon, "Foreign Function Interface (FFI)," 2020.

[52] M. Bouzid, Webpack for Beginners: Your Step-by-Step Guide to Learning Webpack 4, Apress, 2020.

[53] Wasm-bindgen, "Rustwasm," April 2019. [Online]. Available: https://rustwasm.github.io/wasm-bindgen/. [Accessed March 2022].

[54] E. You, "Vue.js - The Progressive JavaScript Framework," 2016. [Online]. Available: https://vuejs.org/guide/introduction.html.

[55] R. Stoenescu, "Quasar Framework," 2018. [Online]. Available: https://quasar.dev/.

[56] World Wide Web Consortium, "HTML Standard," 14 April 2022. [Online]. Available: https://html.spec.whatwg.org/. [Accessed 16 April 2022].

Appendices

Annex 1 – Rust source code used for the creation of the WebAssembly bundle.

```
use js_sys::Promise;
use wasm_bindgen::prelude::*;

// When the `wee_alloc` feature is enabled, this uses `wee_alloc` as the
global
// allocator.
//
// If you don't want to use `wee_alloc`, you can safely delete this.
#[cfg(feature = "wee_alloc")]
#[global_allocator]
static ALLOC: wee_alloc::WeeAlloc = wee_alloc::WeeAlloc::INIT;

#[wasm_bindgen(module = "/js/bindings.js")]
extern "C" {
    async fn js_to_rust_initialize() -> JsValue;
    fn js_to_rust_set_encryption_scheme(scheme: String) -> JsValue;
    fn js_to_rust_setup_context(
        poly_modulus_degree: i32,
        bit_sizes: Vec<i32>,
        bit_size: i32,
        security_level: String,
        precision: i32,
    );
    fn js_to_rust_fast_setup(
        scheme: String,
        security_level: String,
        processing_speed: String,
        precision: i32,
    );
    fn js_to_rust_generate_keys() -> Vec<JsValue>;
    fn js_to_rust_encrypt(array: Vec<i32>, public_key: JsValue) -> JsValue;
    fn js_to_rust_decrypt(array: String, secret_key: JsValue) -> JsValue;
    fn js_to_rust_add_ciphers(cipher_text1: String, cipher_text2: String) ->
JsValue;
    fn js_to_rust_sub_ciphers(cipher_text1: String, cipher_text2: String) ->
JsValue;
    fn js_to_rust_multiply_ciphers(cipher_text1: String, cipher_text2:
String) -> JsValue;
    fn js_to_rust_square_cipher(cipher_text1: String) -> JsValue;
    fn js_to_rust_exponentiate_cipher(cipher_text1: String, power: i32) ->
JsValue;
    fn js_to_rust_negate_cipher(cipher_text1: String) -> JsValue;
    fn js_to_rust_add_plain(cipher_text: String, plain_text: Vec<i32>) ->
JsValue;
    fn js_to_rust_sub_plain(cipher_text: String, plain_text: Vec<i32>) ->
JsValue;
    fn js_to_rust_multiply_plain(cipher_text: String, plain_text: Vec<i32>) -
> JsValue;
    fn js_to_rust_deallocate_context();
```

```
    fn js_to_rust_deallocate_parameters();
    fn js_to_rust_deallocate_seal();
    fn js_to_rust_deallocate_library();

}
#[wasm_bindgen]
pub async fn rust_initialize() -> JsValue {
    let result = js_to_rust_initialize().await;
    result
}
#[wasm_bindgen]
pub fn rust_set_scheme(scheme: String) {
    js_to_rust_set_encryption_scheme(scheme);
}
#[wasm_bindgen]
pub fn rust_setup_context(
    poly_modulus_degree: i32,
    bit_sizes: Vec<i32>,
    bit_size: i32,
    security_level: String,
    precision: i32,
) {
    js_to_rust_setup_context(
        poly_modulus_degree,
        bit_sizes,
        bit_size,
        security_level,
        precision,
    );
}
…
```

Annex 2 – Typescript library implementation source file:

```
import { EasyScheme, EasySecurity, EasySpeed, EasyPrecision } from "./types";
export { EasyScheme, EasySecurity, EasySpeed, EasyPrecision } from "./types";
export type EasyCipherText = CipherText.CipherText;
export type EasyPublicKey = PublicKey.PublicKey;
export type EasySecretKey = SecretKey.SecretKey;
export type EasyContext = Context;
export type EasyEncryptionParameters = EncryptionParameters;
export class Cipher {
    private module: any;

    constructor(module: any) {
        this.module = module;
    }
    /**
     * Method that adds two ciphertexts
     * @param {EasyCipherText} cipherText1
     * @param {EasyCipherText} cipherText2
     * @returns {EasyCipherText}
     */

    add(cipherText1: string, cipherText2: string): EasyCipherText {
```

```
    return this.module.rust_add_ciphers(cipherText1, cipherText2);
}

/**
 * Method that subtracts two ciphertexts
 * @param {string} cipherText1
 * @param {string} cipherText2
 * @returns {EasyCipherText}
 */

sub(cipherText1: string, cipherText2: string): EasyCipherText {
    return this.module.rust_sub_ciphers(cipherText1, cipherText2);
}

/**
 * Method that multiplies two ciphertexts
 * @param {string} cipherText1
 * @param {string} cipherText2
 * @returns {EasyCipherText}
 */

multiply(cipherText1: string, cipherText2: string): EasyCipherText {
    return this.module.rust_multiply_ciphers(cipherText1, cipherText2);
}

/**
 * Method that squares a ciphertext
 * @param {string} cipherText
 * @returns {EasyCipherText}
 */

square(cipherText: string): EasyCipherText {
    return this.module.rust_square_cipher(cipherText);
}

/**
 * Method that exponentatiates a ciphertext to a certain power
 * @param {string} cipherText
 * @param {number} power
 * @returns {EasyCipherText}
 */

exponentiate(cipherText: string, power: number): EasyCipherText {
    return this.module.rust_exponentiate_cipher(cipherText, power);
}

/**
 * Method that inverts all the values of a ciphertext
 * @param {string} cipherText
 * @returns {EasyCipherText}
 */

negate(cipherText: string): EasyCipherText {
    return this.module.rust_negate_cipher(cipherText);
}
```

```
    /**
     * Method that deallocates the wasm module reference
     */
    delete() {
        this.module = null;
    }
}

export class Setup {
    private module: any;
    constructor(module: any) {
        this.module = module;
    }
    /**
     * Method that will initialize the bindings between EasyFHE, SEAL and
node-seal
     * @returns
     */
    initialize(): Promise<FHEModule> {
        return this.module.rust_initialize();
    }

    /**
     * Method that sets the  security Context of the module
     * @param {number} poly_modulus_degree
     * @param {Int32Array} bit_sizes
     * @param {number} bit_size
     * @param {EasySecurity} security
     * @param { EasyPrecision } precision - precision in bits (only used for
CKKS scheme)
     */
    setContext(
        polyModulusDegree: number,
        bitSizes: Int32Array,
        bitSize: number,
        security: EasySecurity,
        precision = EasyPrecision.NORMAL
    ): void {
        this.module.rust_setup_context(polyModulusDegree, bitSizes, bitSize,
security, precision);
    }

    /**
     * Method that will do the setup of the module in a very simplified way.
     * @param { 'bfv' | 'bgv' | 'ckks' } scheme - homomorphic scheme used
     * @param { EasySecurity } security - security measured in bits
     * @param { EasySpeed } processingSpeed - refers to the size of
polymodulus degree, the greater the degree, the heavier the computational
cost will be
     * @param { EasyPrecision } precision - precision in bits (only used for
CKKS scheme)
     */
    fastSetup(
        scheme: EasyScheme,
        security: EasySecurity,
        processingSpeed: EasySpeed,
```

```
        precision = EasyPrecision.NORMAL
    ): void {
        this.module.rust_fast_setup(scheme, security, processingSpeed,
precision);
    }

    /**
     * Method that deallocates the wasm module reference
     */
    delete() {
        this.module = null;
    }
}
```

Annex 3 – Usage of the module in a Quasar application:

```
<script lang='ts'>
import getFheModule, {
  EasyScheme,
  EasySecurity,
  EasySpeed,
  FHEModule,
} from 'EasyFHE';
(async () => {
    // Get the WebAssembly module
    const EasyFHE: FHEModule = await getFheModule();
    // Initialize it
    await EasyFHE .Setup.initialize();
    // Set your preferred encryption parameters
    EasyFHE .Setup.fastSetup(
      EasyScheme.BFV,
      EasySecurity.TC128,
      EasySpeed.NORMAL,
    );
    // Generate a keypair
     const [publicKey, secretKey] = EasyFHE .generateKeys();
     // ciphertext * plaintext
     //--------------------------------------
     const leftValue0 = EasyFHE .encrypt(Int32Array.from([23,4,5]),
publicKey);
     const rightValue0 = Int32Array.from([2,3,54,2]);
     const encResult0 = EasyFHE .Plain.multiply(leftValue0.save(),
rightValue0);
     const result0 = EasyFHE .decrypt(encResult0.save(), secretKey);
     //--------------------------------------
    EasyFHE .deallocateLibrary();
    publicKey.delete();
    secretKey.delete();
})()
</style>
```